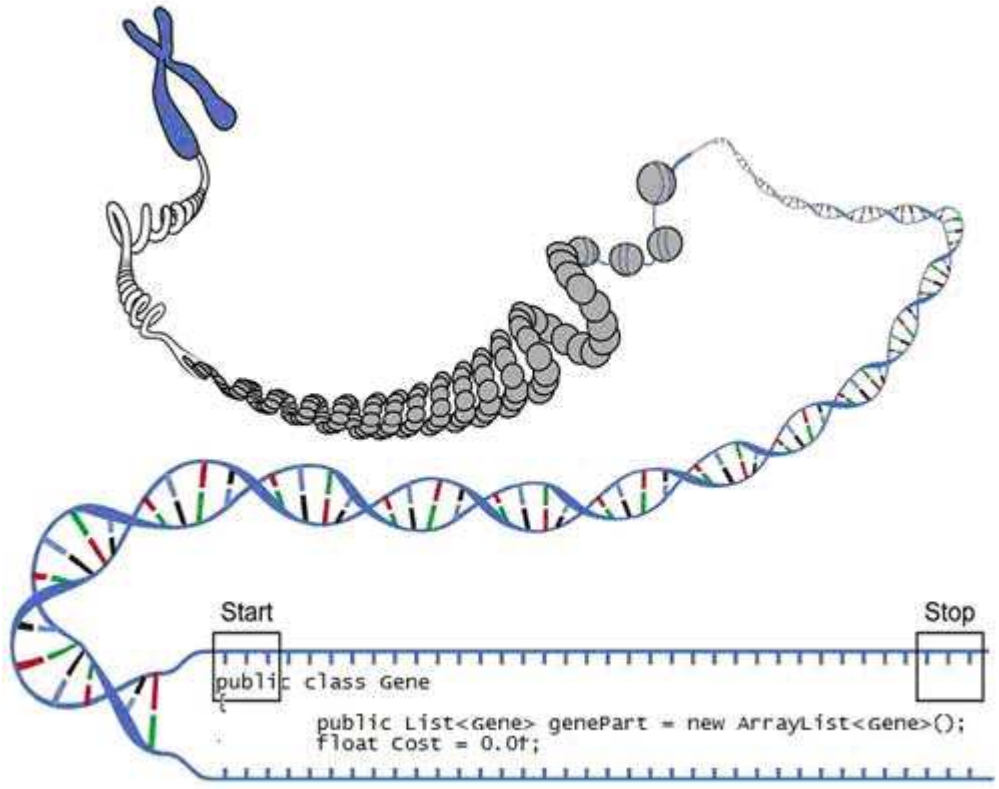
The use of Genetic Algorithms on The Bin Packing Problem.



By

Jakob Just Sørensen 19-09-1977

and

Henning Mortensen 29-07-1974

Supervisor

Yvonne Dittrich, assoc. prof

Table of Contents

Subject

Usage of Genetic Algorithms

Genetic Algorithms (GAs) are already being used in everyday tools like scheduling software to optimize the usage of resources[1] and in computer games to balance the game to fit the player's style and expertise level.[2] GAs have even been successfully applied to the field of designing analog circuits[3] and researchers at Georgia Tech Center for Music Technology have succeed in creating robots that can analyze and play improvised music real-time along with humans who are playing other instruments.[4]

Reason for selecting the subject.

Most of the programming done today has a very specific job to solve and often breaks if presented with parameters outside of the predefined solution scope. The idea of programming in a way that the code is able to evaluate a solution (or itself), could very possible be the future of programming and also one of the basic steps towards artificial intelligence and Singularity[5].

This "Brave New World" of programming was our interest point and introduction to the subject.

But if GAs holds such potential, why is it not more commonly used? This is one of the questions which we will try to look more closely into in this report with the basis in our own implementation of Genetic Algorithms – The Bin Packing Problem (BBP).

Applying GAs to The Bin Packing Problem.

The Bin Packing Problem revolves around the issue of fitting a number of boxes into a larger bin. The problem is easily defined but very hard to solve. In math terms the problem is called NP-Hard[6], which means that given a set of boxes and a bin, it is very hard to answer questions like: what is the best way of packing the bin, what an optimal solution would be and how good it would be. The only way of finding the optimal solution is to use the brute force technique of trying every possible packing, which often will take too much time or being close to impossible to do because of the number of possible combinations.

This is where GAs come in: By improving upon a solution over and over again, in the same way as *evolution has allow us to evolve from single cell creatures to what we are today,* GAs will gradually move towards a better and better solution.

The complexity increases drastically the more factors you have to take into consideration when doing an implementation like the BBP; therefore we have chosen to do a simplified version of the BPP where we will only be working in two dimensions and a single bin. The goal is not to find an optimal solution, but to find good approximated solutions within a short time (minutes) using a GA.

---

[1] http://www.orchestrate-plan.com/orchestrate_optimizer.htm
[2] Cole , Nicholas, and Louis, Sushil J. and Miles, Chris "Using a Genetic Algorithm to Tune First-Person Shooter Bots" 2004 - University of Navada
[3] Johson , R. Colin "Genetic program auto-designs anlog circuits" – Issue 904, 2003 - Publisher: eeTimes
[4] http://blog.wired.com/music/2008/11/no-way-robot-ja.html
[5] http://en.wikipedia.org/wiki/Technological_singularity
[6] Non-deterministic Polynomial time hard (http://en.wikipedia.org/wiki/NP-hard)

Boundaries.

There are many closely related programming styles that resemble and often contain GAs such as: Genetic programming (GP), Evolutionary programming, Evolution strategy, etc. Common for them all are that they relay on the principals of *biological evolution* to optimize, approximate or handle problems. It would be out of the scope of this report to explain or even try to draw parallels between these and GAs, but in certain areas of the report there can be elements of these that we will refer to.

GAs also embodies a wide variety of genetic operators, we described some of the most common ones in the introduction to Genetic Algorithms, but we will not be including or explaining other types then the ones used in our implementation.

Method, structure and sources.

Using the model illustrated in figure 1[7], we aim to build upon our existing knowledge about software development to increase our understanding of GAs and how it can be applied. In this report the Research Domain is GAs and the methodologies are a combination of prototyping (our implementation), discussion/observation (literature describing real world usage of GA) and test results (output from our implementation). The implementation will be central focal point of the report, it will be used both to test the idea behind GAs and try to determine if the usefulness of GAs as a conventional programming tool.

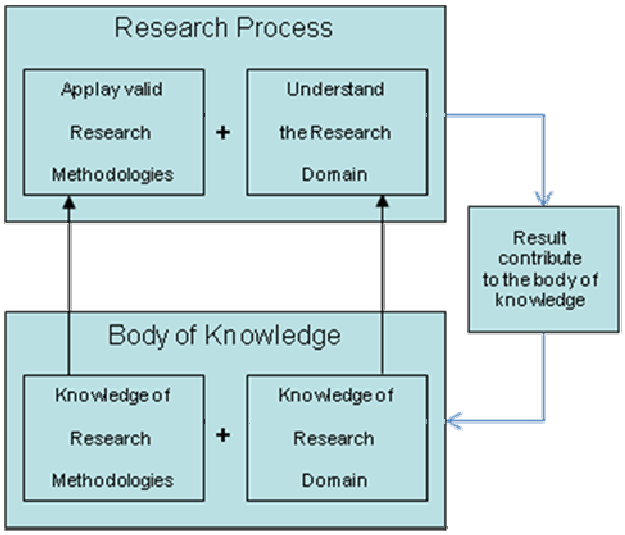


Figure 1 – A framework of Research

We will begin with a introduction to GAs, then we will discuss the technical and business related issues concerning the use of genetic algorithms, followed by our implementation of the bin packing problem. Based on the findings from our implementation we will try to draw a conclusion about the usefulness of GAs for this type of problem.

The majority of the sources that we reference are published books, university papers or articles in well known and established magazines, so the validity of these is to the best of your knowledge of a standard that can not be drawn into question.

The links we reference are selected based on credibility, this is done by evaluating the article itself and insuring that the link is referenced from more then one source, we also verify that the information on the internet page can be found at other sources. In some cases we use Wikipedia as a source, we are aware of the fact that Wikipedia is an open source encyclopedia that everyone can edit and that this raises a question of it credibility, but the primary use of Wikipedia in this report is of technical or mathematical heavy articles that is based on derived facts rather then subjective evaluation.

[7] Nunamaker, Jay F. "Journal of Management Information Systems" 1991, page 92

Introduction to Genetic Algorithms.

Genetic algorithms are part of a programming discipline which uses the *principals of evolution* to find an exact or approximated solution to a problem. GAs are a direct metaphor for having code react and evolve in the same way as *biological evolution* or trying to mimic this. GAs are search based and incorporates natural selection to refine a solution over several generations/iterations. First suggested by Alan Turing in 1948 and pioneered by John Holland in the early 70s,[8] it has since been widely studied and used to produce results capable of matching, and in few cases exceed, those of a more conventional approach.[9]

The basic ideas behind GAs are: There is a *genetic pool* that potentially contains a solution, or a better solution, to a given problem. The solutions are represented as a *chromosome*, which consists of *one or more genes*. Each *gene* represents an action that needs to be executed; the actions can be everything from mathematical expressions, orderings and transformations to whatever can be applied to the problem at hand. The potential solutions are found by taking an initial random population of chromosomes as possible solutions, then applying genetic operators such as mutation and crossover to *evolve* them over numerous generations.

Each generation normally contains more then one chromosome (solution), a fitness function is then applied to each chromosome to evaluate it and decide if it is a better solution. Once evaluated a new population of *chromosomes* are generated from the previous batch and the genetic operators are applied. This process is repeated until an acceptable solution is found or the maximum number of generations has been reached. The reason for restricting the number of iterations can be many; it can be because of time restrictions, it can be for statistical reasons, etc.

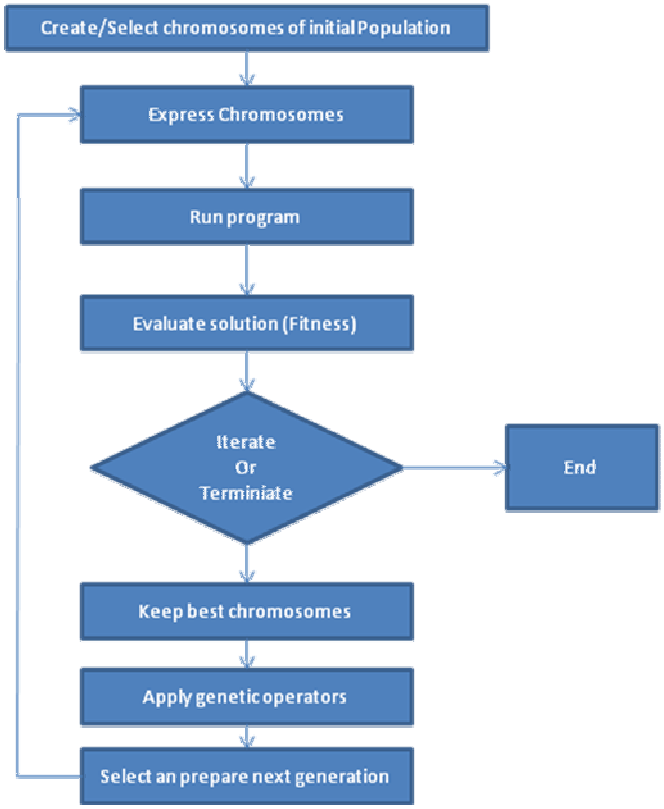


Figure 2 - the normal flow for genetic algorithms.

There are many different genetic operators, but the most common are:

Selection – A selection of *chromosomes* from the current population based on randomness, but often includes a fitness function to insure that the best *genes* are more likely to survive.

[8] Poli , Riccardo and Langdon, William B. and McPhee, Nicholas Freitag. "A Field Guide to Genetic Programming" 2006 - Publisher: Lulu.com - Page 141

[9] http://www.genetic-programming.com/humancompetitive.html

Mutation – Mutation alters one or more of the gene values in a *chromosome* from its initial state. The importance of this is that is helps to insure diversity in the population, which else could become too similar and causing the evolution to stop.

Crossover – Crossover combines *two chromosomes* to produce *one or two new chromosomes*, which may be new and better *chromosomes*.

GAs are especially useful when working with search and optimization, where the problem is easily defined and the number of possible solutions exceed what is reasonable to test via brut force techniques within a acceptable time span.

One of the most commonly used problems when discussing GA is The travelling sales man problem (TSP)[10], which consists of the problem of a sales man who needs to visit a number of cities. How does he order the visits to the cities so he only visits each city once, but also take the shorts possible route? We will throughout this report be referring to the TSP as the common sample problem. The reason this problem fit the boundaries of GA so well is because it works with a search based criteria that otherwise would be hard to express using traditional programming techniques, or which would require that you have to go through all possible solutions. Using GAs you would be able to quickly get a solution that in most cases would be sufficient, but it is important to remember that GAs in most cases only give approximated solutions and not the optimal one(s).

[10] Hauot , Randy L. and Haupt, Sue Ellen "Practical Genetic Algorithms" 2004 - Publisher: Wiley-Interscience - Page 124

Technical issues.

The complexity of the BPP is of such a character that using a GA is a good alternative to the traditional brute force search, but the complexity is not only in the problem to be solved it can also be in the solution found by the GA.

Complexity.

The complexity of the BPP is mind buckling. When the number of packages are low the problem is fairly simple, but the complexity of the problem grows exponential with the number of packages. For instance the numbers of possible combinations for 15 packages that can be oriented in 2 directions are

$$(15 \cdot 2) \cdot (14 \cdot 2) \ldots (1 \cdot 2) = 15! \cdot 2^{15} = 42849873690624000 .$$

At first we have 15 packages to select from and place in the bin in 2 orientations ($15 \cdot 2$), then we 14 packages left to select from and place in the bin in 2 orientations ($14 \cdot 2$) and so on. If we did the same calculations in 3d the number of possible combinations would be

$$(15 \cdot 6) \cdot (14 \cdot 6) \ldots (1 \cdot 6) = 15! \cdot 6^{15} = 6148488525485105479680000$$

or 3^{15} times the number of combinations for the 2d version. This shows without a doubt that the complexity of the BPP is huge and also the reason that we only work in 2d on our implementation.

CPU time.

A brute force test on this "small" bin packaging problem using only 15 packages in 2d, would require a computer capable of checking a million packing combinations a second and it would still take more than 10 years to complete. But since GAs uses evolution to improve the solution, it will rather quickly start to improve and become increasingly better within a short time (a few minutes), even with a relatively small computer.

Time required to analyze the results.

GAs will often give solutions that are very complex, so depending on the usage you will often have to do some analysis of the solutions to make sure they perform the desired task. Some GAs generates solutions which will perform, but also contains "garbage" information[11] which have no impact on the solution, so removing this redundant code would improve performance. This is also seen in real life where as much as *95% of the human genome is junk DNA*[12].

Lack of conversion

The GA does not necessarily give a solution that will work. You have to design your fitness function with great care, as this is essentially the only task you are giving the GA to solve, or in other terms you only way

[11] Ferreira, Cândida "Gene Expression Programming: A New Adaptive - Algorithm for Solving Problems" Universidade dos Açores, Portugal

[12] http://en.wikipedia.org/wiki/Junk_DNA

of evaluation a possible solution.
The GA will try to solve the task with the tools provided, like selection, mutation and crossover, but this in certain cases is not enough to ensure the GA will converge. Sometimes, especially if there are many factors that it needs to take into consideration, the GAs will run into an *evolutionary dead-end* and stop evolving.

Lack of understandable patterns.

The complexity of some GA solutions makes it a daunting task to understand the patterns within it. As in the example of a program evolved by Jaime J. Fernandez to pick up signals from 3 electrodes on a person's wrist[13], to show which way the person moved his/her thumb, the solution was one long line of hundreds of nested "if statements". It worked perfectly but the code was to complex to give any indication on why it worked.

Errors in the genes

In contrast to *biological evolution* where errors in the genes might not have an effect, in GAs errors will often be fatal for the program because of the way a computer works. So even though they build on the same principals, there are very important areas where they differ and the idea behind Genetic Programming will properly not be fully achieved before the computer itself is able to adapt in the same way as the code.

Sub conclusion A.

GAs are good alternatives to brute force when you take into account the limitations of the GAs. You should always ensure the GAs only does what it is suppose to do and it will often even be an good idea to put restrictions on the output of the GA. E.g. if you make a program to control an artificial arm, you want to make sure that the arm in no way, can hurt the person it is attached to.

GAs can solve most problems, especially if they are search or optimization based, you just have to convert the problem into a structure a GA can be applied to and give it at a good goal in the form of a fitness function. But you always need to remember that it is a computer that is handling the problem, and computers are build on the principals of yes and no – 0 and 1 – on and off. So problems are handled digital not analog, this can sometimes pose a problem for a theory that is based on living cells.

[13] Gibbs , W. Wayt "Programming with Primordial Ooze" – October, 1996 - Publisher: Scientific American

Business issues.

The BPP is a problem that many companies around the world encounter on a daily basis. If they send a large number of packages, then a program for optimizing that packing could be of great financial benefit for them, especially if the cost of transportation is high.

Seldom gives a final solution, but only a refined guess.

An important factor in using GAs is that they almost never give the optimal solution to a problem, but what they do give is an approximated solution. If we look at one of the results from our BPP implementation – Figure 3– we clearly see that the waste percentage slowly drops with each generation (the unit for waste is 1 = 100%). After 5000 generations the waste factor is around 8% of the bin.

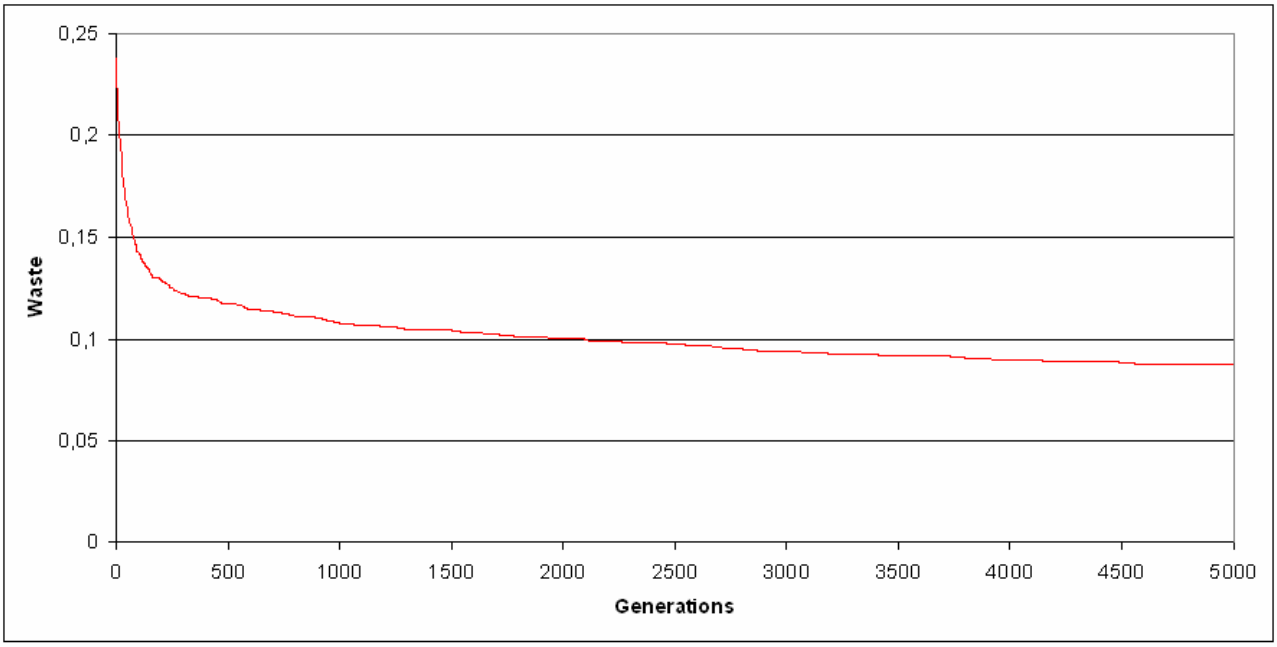


Figure 3 - one run of our BPP

Depending on the problem type, the approximation may vary and in many cases the solution is only acceptable if it is within a certain range.

Often a fragile solution, only works for one specific problem.

A problem that can be of great importance depending on the type of problem you are working with, is the fragility of the solutions that a GA comes up with. If you change any part of the solution, then you need to run the program again, and it is very unlikely that it will come up with a solution that resembles the original solution. Again, if we look at the BPP, if the dimensions of any of the boxes are changed, then it would require a new run, which is very unlikely to look anything like the first one.

Often needs to be refined.

GAs are not intelligent and work to some extent by the trail and error concept, so what may be easily spotted by a human might be impossible for GAs. Figure 4 below show an example of a packing done by our

implementation of the BBP, it is easy to see that by moving the last small box to one of the spaces above, the packing would be much more efficient.

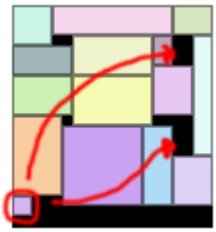


Figure 4 – human refinement of a solution

This human touch is often very important for successful usage of GAs and computer programs in general. In the famous rematch between Gary Kasparov and IBM's Deep Blue chess computer in 1997, they where tied after the first 4 matches, but then something happened and Deep Blue suddenly started to play more intelligently according to Kasparov. As part of the rules for the competition, IBM was allowed to adjust the program between matches and by doing this they achieved the first victory in a competition over Kasparov[14].

Some would argue that the need for this human touch is just shortcomings in the programming, that since it did not take these factors into account the program was build in the wrong way.

Development time

Time is often a factor in developing programs, in particular when working with GAs. As discussed in the "Technical Issues", the complexity in both developing the program and in understanding the solution can be of such a character that the time spend on this becomes too high. This is especially true for GAs if there to many factors that it needs to take into consideration.

Sub conclusion B.

GAs holds great potential for solving complex search based problems, this is not only confined to the BPP but everywhere a approximated solution is acceptable. Especially in areas of resource and scheduling planning could the usage of GAs be used with great success. But for many years to come there will be a need for the human touch, this small push in the right direction, to insure the solution performers to the best of its capabilities.

[14] http://en.wikipedia.org/wiki/Kasparov#Deep_Blue.2C_1997

Our implementation

Our implementation is a GA written in Java in an effort to solve the BPP.

Reason for doing the implementation

We did the implementation to show that it is possible to obtain knowledge to implement a GA in a short amount of time and without previous programming experience within the area. The idea behind GA is very simple but powerful and this is what we wanted to show.

Input for the BPP

A list, containing information about the height and width of the boxes.
Information about the bin's dimensions.

Constraints on the BPP

No package must be wider or taller, than the width of the bin.
The generation size has to be larger than 2
Number of generation has to be larger than 1

Expected output from the BPP

The output should as minimum, be a list of the packages in the order to be packed, the resulting percentage waste and a picture to show the packaging.

The Architecture[15]

The Gene Class holds information about a single box, a floating point number is used to determine its packaging order in the bin, and a Boolean to indicate if the box should be rotated or not.

Class Chromosome contains a list of Genes from the gene class. A chromosome describes an entire packaging of a bin.

The Evolver class uses a list of chromosomes, to represent an entire generation. These are evolved into a new generation by the evolution method within the Evolver class. This is done until an acceptable solution is found or the maximum generations have been reached.

[15] A UML diagram showing the layout of the implementation can be found on the CD.

Evolver()

The call to the Evolver method, Figure 5, starts the GA on the search for the best packing of the boxes given.
First the total area of the boxes are calculated, this value is later used in the fitness calculation. Then a set of total random chromosomes are generated. The chromosomes are then prepared for the fitness method by inserting the box numbers in the chromosomes in order. The fitness method is then called which calculates and sets the fitness of the chromosome. SortChromosomes are called to sort all the chromosomes so the best chromosomes are first in the list. Finally the Evolution method is called, this method generates a new generation from the current generation of chromosomes.

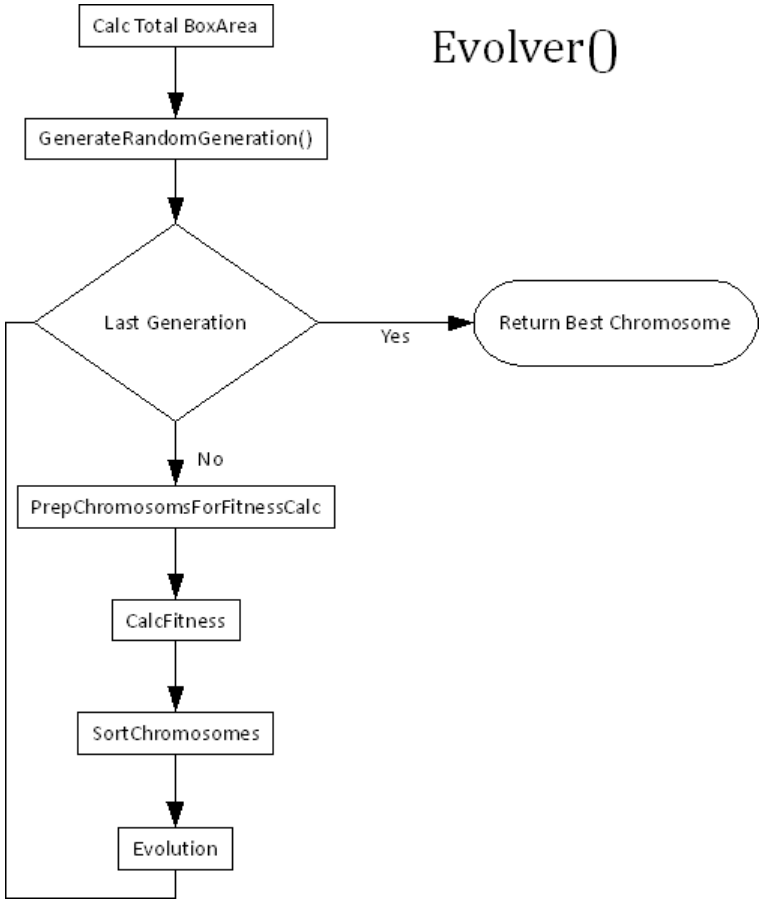


Figure 5 – Evolver flowchart

CalcFitness()

The CalcFitness method calculates the fitness of a chromosome and set the fitness field of that chromosome.
First a copy is made of the chromosome, as the genes in the chromosome are sorted in this method. Then the genes in the copy are sorted giving the packing order.
Then the chromosome is passed to the bin packing method. This method does the actual evaluation of the packing described in the chromosome. The result is returned and stored in the original chromosome.

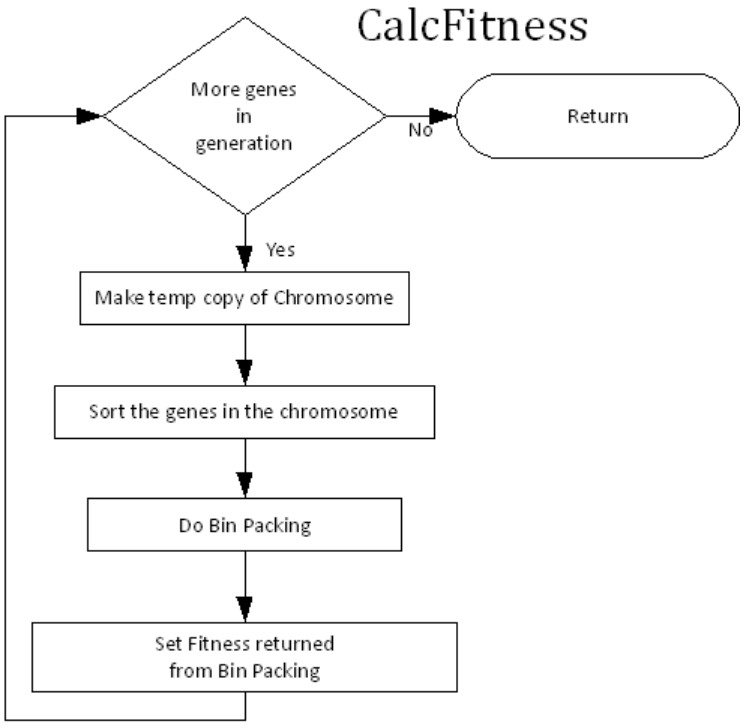


Figure 6 – CalcFitness flowchart

Evolution()

The Evolution is the method that does the actual evolution from generation to generation. First a new (empty) list of chromosomes is made. Then two chromosomes are selected from the old generation and they are combined into two new chromosomes by crossover, mutation and geneswap. This is done until the new generation holds the same number of chromosomes as the old generation.
The selection of the chromosomes from the old generation is done so the chromosomes with the best fitness are more likely to be selected than the chromosomes with a poor fitness. The selection also ensures not to select the same two chromosomes for *mating*.
A pair of chromosomes can be subjected to the 3 different kinds of operations. But as the operators are selected by chance, a pair of chromosomes can be subjected to everything from zero to all three operators.

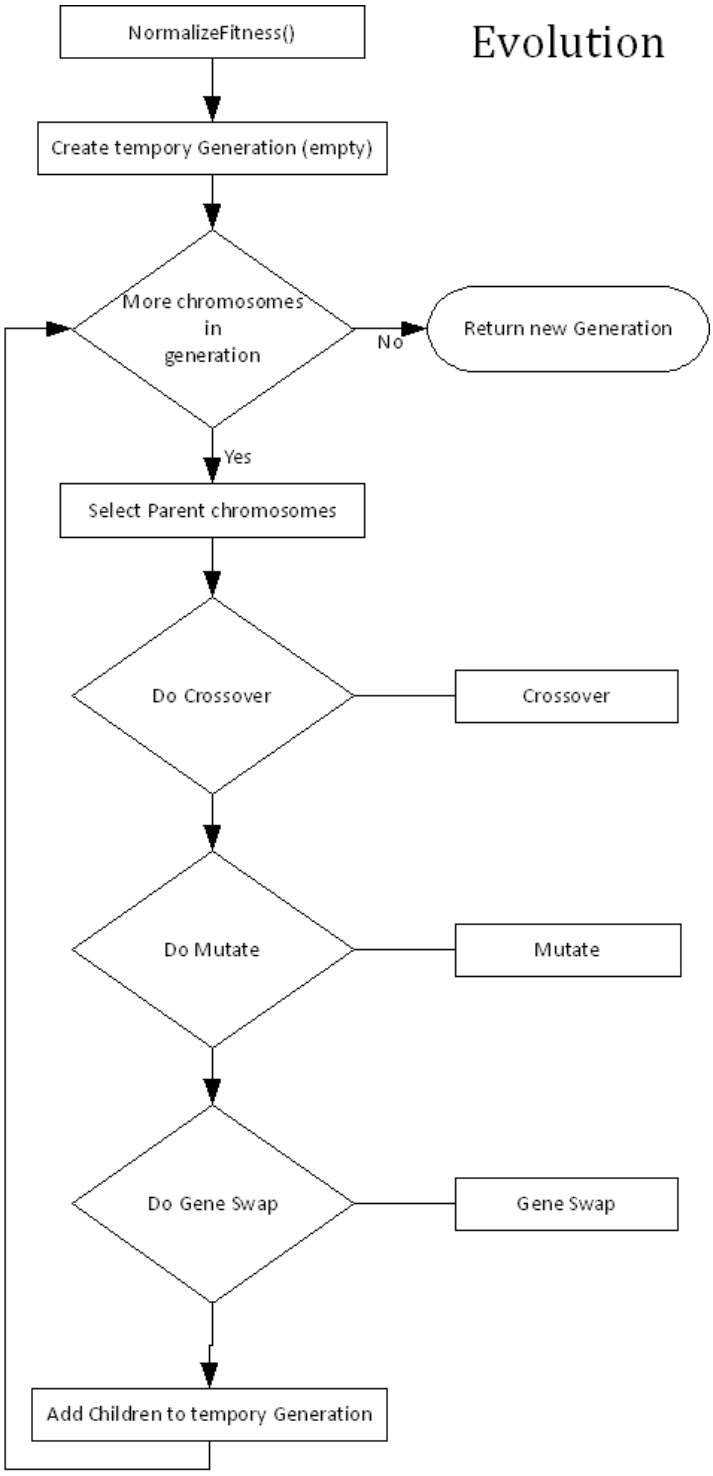


Figure 7 – Evolution Flowchart

The genetic operators

Selection

The selection of the chromosomes from the old generation to be mated, are weighted so the better the fitness the more likely they are to be selected for *mating.* In Figure 8 shows the weighting function used, the function are made so the top 25% of the chromosomes are as likely to be selected as the remaining 75%

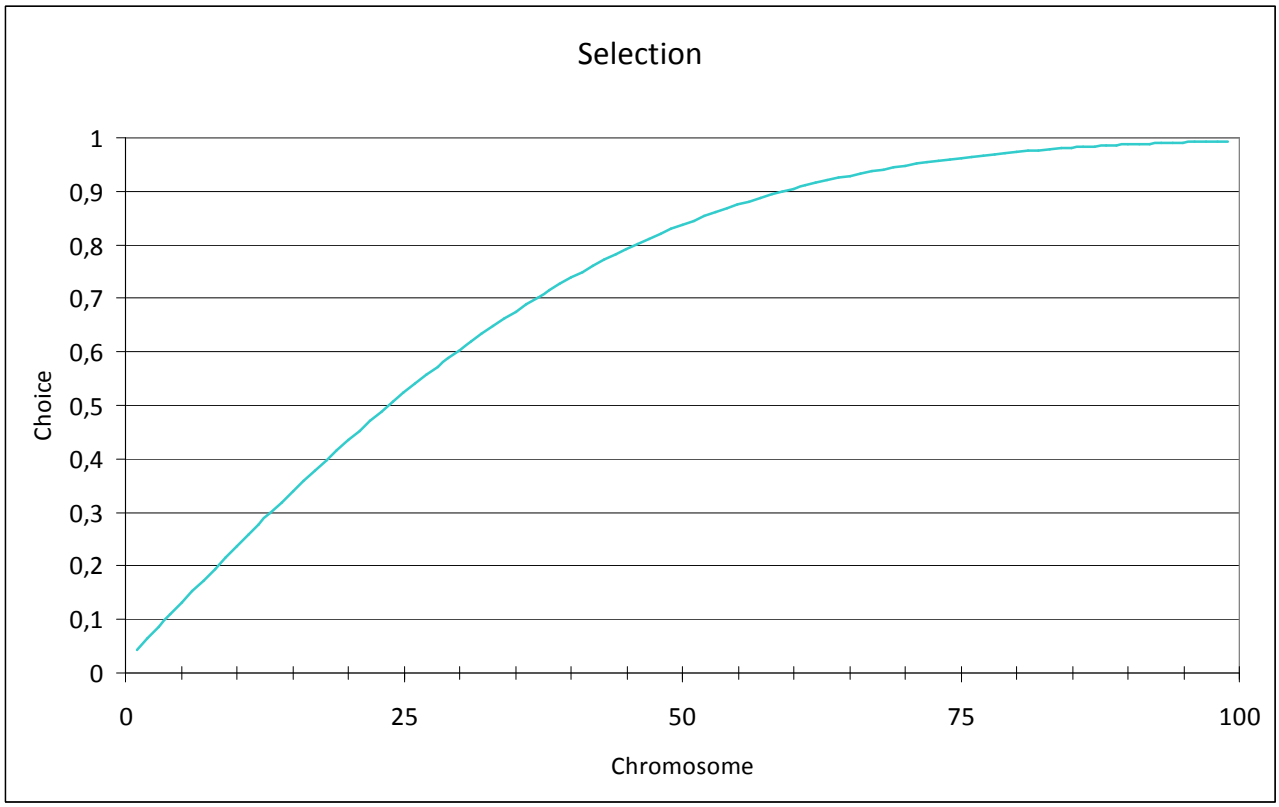


Figure 8 – Selection

The weighting is made from a combination, of the chromosomes location in the sorted list and a mathematical function selected purely to give the desired function in Figure 8.

Mutation

The mutation operation takes a chromosome and selects a random gene within it, and then selects one of the two features represented in the gene to be modified. Once one of the two features are selected, it is overwritten by a random value.

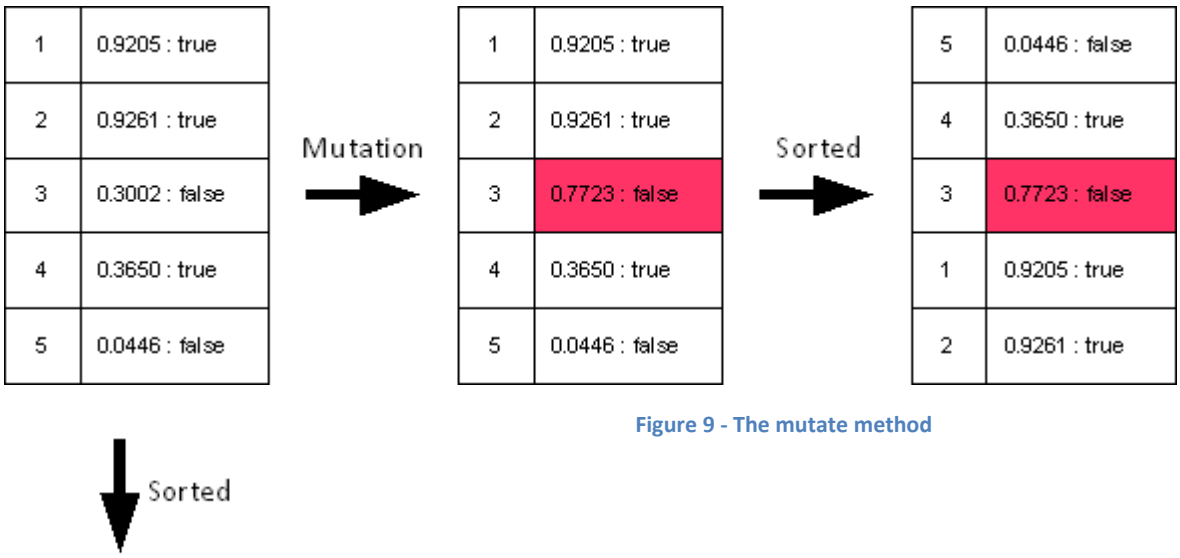


Figure 9 - The mutate method

In the example shown, the selected feature is the value representing the location of box 3. The value are changed from 0.3302 to 0.7723, this changes the packaging so box 3 now comes before box 4.
The ordering of the packages are by the mutation method changed from 5,3,4,1,2 to 5,4,3,1,2 as shown, so most of the ordering are kept, which is what we want from a mutation method.
The mutation could also have been to the feature representing how the rotating of one of the boxes should be. The ordering would then be the same, but the box (gene) could be rotated.

Crossover

The crossover method should combine two chromosomes by taking parts of one and swapping it with another. It should still keeping as much information from the two as possible.

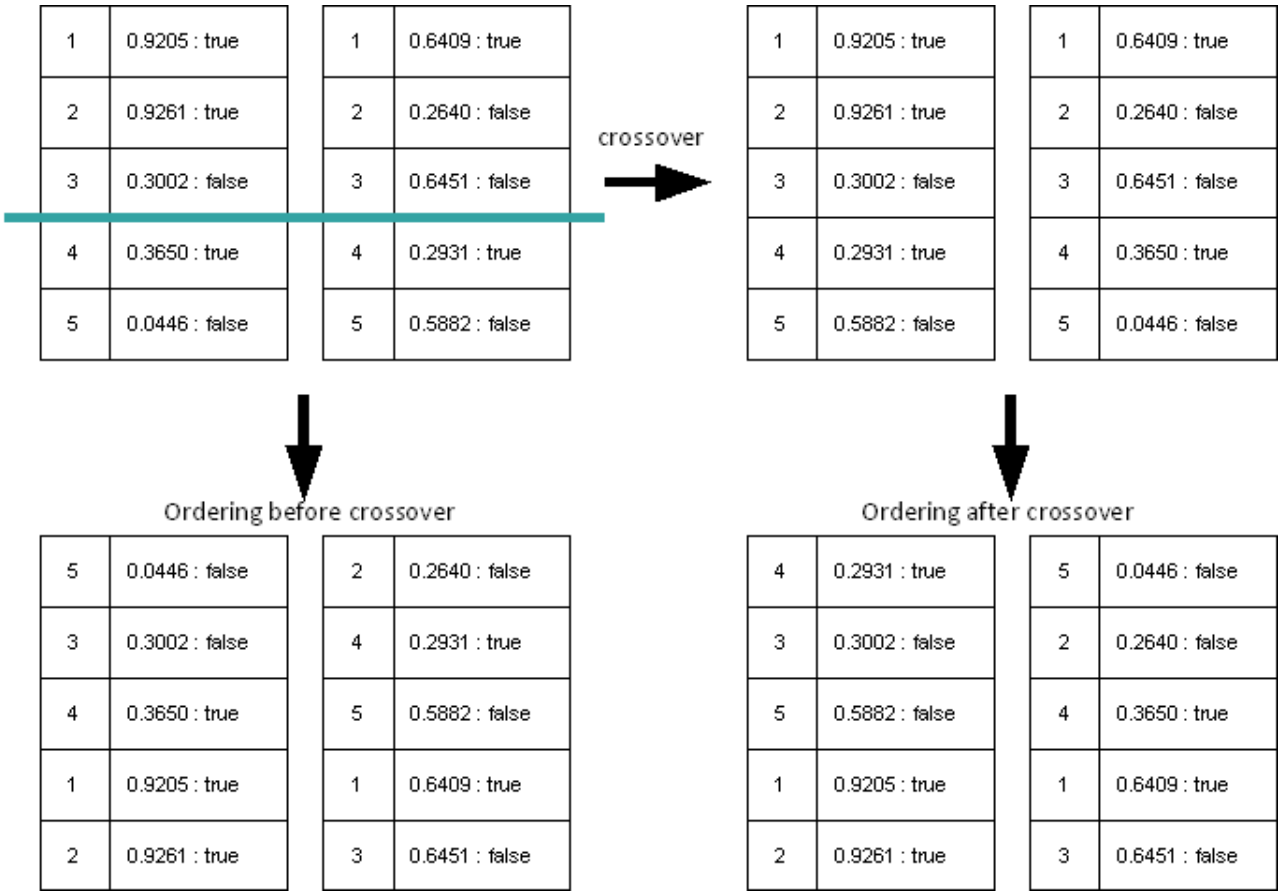


Figure 10 – The crossover method

The example in Figure 10 shows how the crossover method works on our chromosomes.
The first chromosome has after the crossover swapped gene 4 and 5, and the second chromosome has permutated gene 2,4,5 into 5,2,4. So as expected of a crossover function, some of the information hold in the chromosome is preserved.

How does our GA work

The GA, performs two tasks. Find the order in which to place packages in the bin, and in what orientation. The bin packaging routine only need these two information's together with a list of packages, to do an evaluation of the packaging. The packaging routine is explained below in Figure 12.

A list of packages, in the order and orientation to be packed are shown in Figure 11.

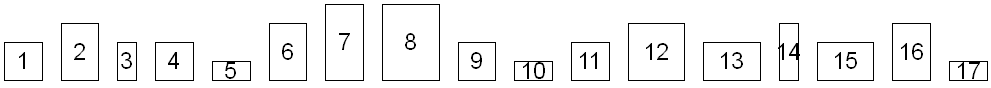


Figure 11 – List of boxes

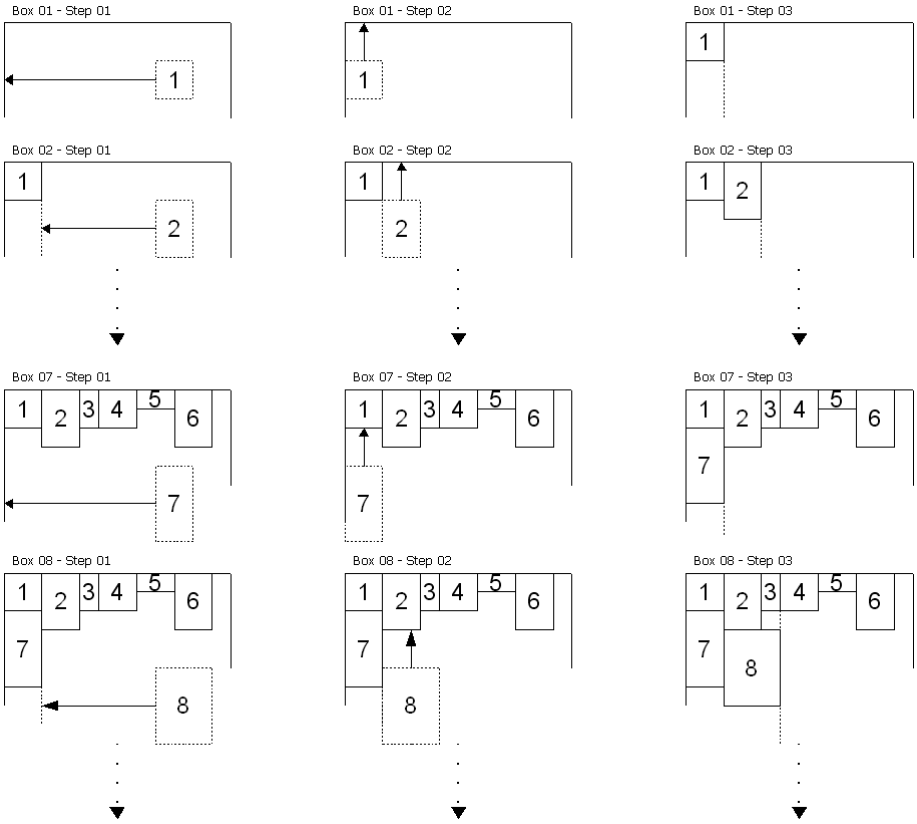


Figure 12 – Packing the bin

The cost for each chromosome is calculated as the space wasted in the bin, after all the packages are placed in the bin.

$$\text{cost} = 1 - \frac{a}{y \cdot x}$$

Where y is the highest point of the top most package in the bin, x is the bin width, a is the total area of the packages. So an optimal solution would have a cost of zero, as the area of the bin is equal to the area of the boxes.

The final packing is shown in Figure 13, we find a cost of 0,348 from the values $y = 55, x = 60, a = 2150$. This means that 34,8% of the space in the bin is wasted.

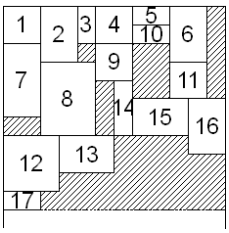


Figure 13 – A packed bin

The travelling salesman problem, a different approach.

TSP is the problem of a salesman who needs to find the shortest route between a number of cities.

When representing the route as an ordered list of cities it gives a problem when doing the crossover function. The problem is that: if you use an ordinary crossover function you will end up with the same city represented more than once, or not at all in the chromosome as shown in Figure 14.

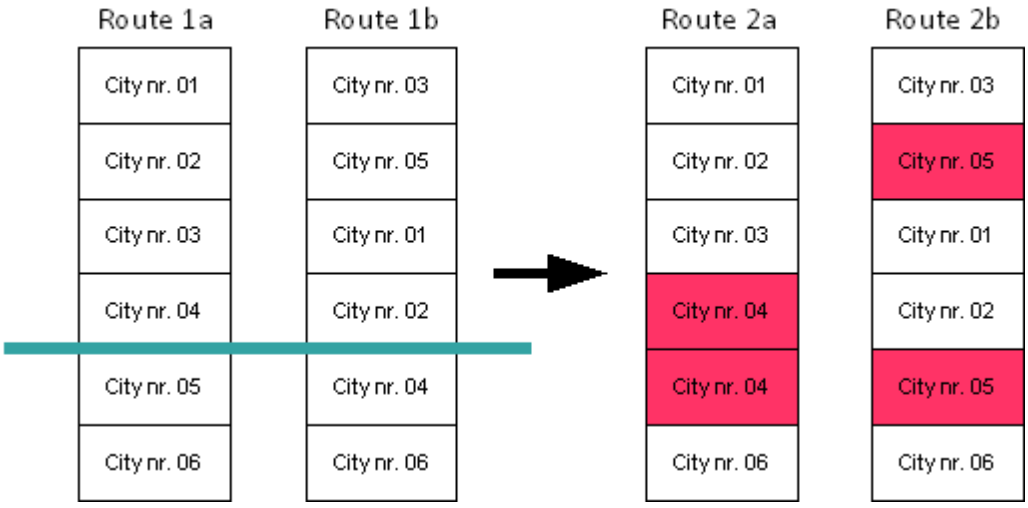


Figure 14 – TSP crossover problem

There are ways to avoid this. In the book Practical Genetic Algorithms, they show multiple ways of changing the crossover function to circumvent the problem. We did not want to change the algorithm so instead we changed the mark up of our chromosome so we could use the "standard" crossover function. The way we have solve it has to the best of our knowledge never been done before. Since we were forced to begin on parts of the implementation before reading all of the literature, it resulted in us using float values to representing the order of the packages.

Known problems/weaknesses with our implementation.

Giving the algorithm a lot of long upright packages, will result in a bad bin packing as the GA will find it difficult to find good solutions. This is because it would require a lot of packages being rotated in the same generation and the GA (as it is) can only rotate one package in a chromosome at the time. The probability that these rotations would survive from generation to generation and accumulate into a whole row of packages being rotated, are very small.

Output

The output from our implementation is a list containing two columns, one with the best fitness of the current generation, and one with the average of the fitness of all the chromosomes in that generation e.g.

Best Fitness	Average Fitness
0.16667	0.38528
0.16667	0.36524
0.16667	0.36423
0.16667	0.36213
.	.
.	.
and so on.	

Notice that the best fitness rarely changes compared to the average fitness's. This is do to the fact that many generations can go by, without the GA finding a better chromosome than the one currently evaluated to be the best. This is only true as long as Elitism is turned on.

This data can easily be converted to a graph as this:

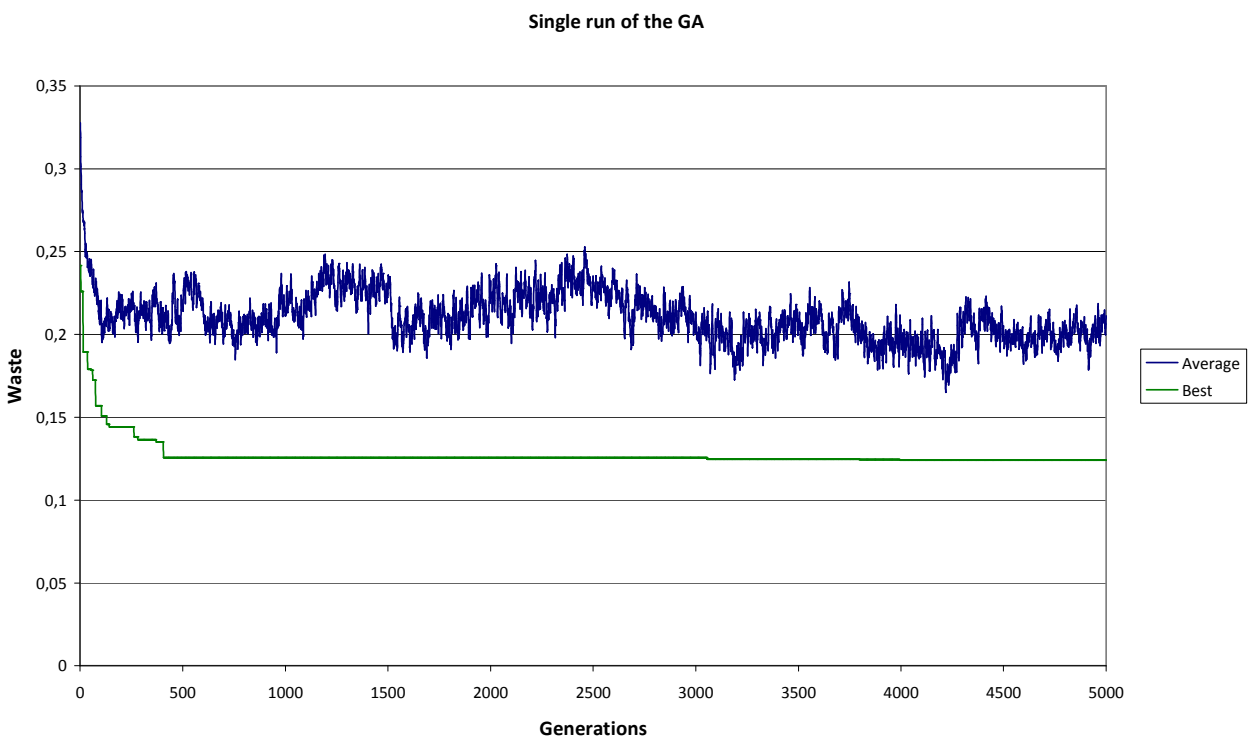


Figure 15 – Graph of a single run of the GA

You will always get a summery of how the resulting packaging is done and a graphics representation of the packed bin (Figure 16 - A packed bin).

Fitness = 4,76%

16	true
8	false
17	false
15	true
7	true
4	false
11	true
10	false
9	true
5	false
0	true
1	true
2	false
3	false
6	false
12	false
14	false
13	false

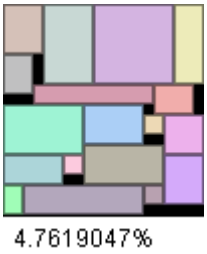


Figure 16 - A packed bin

First column is the package, the second column tells if the package is to be rotated or not.

Suggestions to improve the GA.

1. Insert random chromosomes every generation would result in a bigger search area, allowing the GA to more quickly find solutions outside the search area defined by the first chromosome selection. Though some would argue it would just result in a more random search.

2. Extending elitism to copy more than the best gene, for instance top 5 or 10. This, we believe would give a faster conversion. But would also lower the diversity in the collection of the chromosomes.

3. Evaluate multiple generations and select the best generation NOT the best individual. This could lead to a better chance of finding an optimal solution, as it would make the average fitness in the collection of chromosomes better over generations.

4. Remove identical chromosomes. This would ensure that we do not end up with a lot of identical chromosomes, and thereby loosing diversity in the chromosomes.

GA is not random search or brute force.

If you take the BPP with 15 packages as an example, and compare the chance of finding a better solution with a random search compared to GA, you will find that the GA will be better.
Let's say we use a generation size of 100 and run the GA for 5000 generations this would be a total of 500.000 random ways of packing the bin. And out of the 4284987369062400, it is only about $1{,}167 \cdot 10^{-8}$ % or close to 0%. The interesting thing is that with GAs you do not hit more, but the ones you do hit will be evaluated and the best ones will be improved upon.

Testing and Results.

We will try to answer some obvious questions that arose while doing the implementation. We will do this by running our GA with different settings of the parameters for the GA, and with different packaging files.

1. **Will the GA converge?**
   We did a few runs with a random set of boxes and got this graph

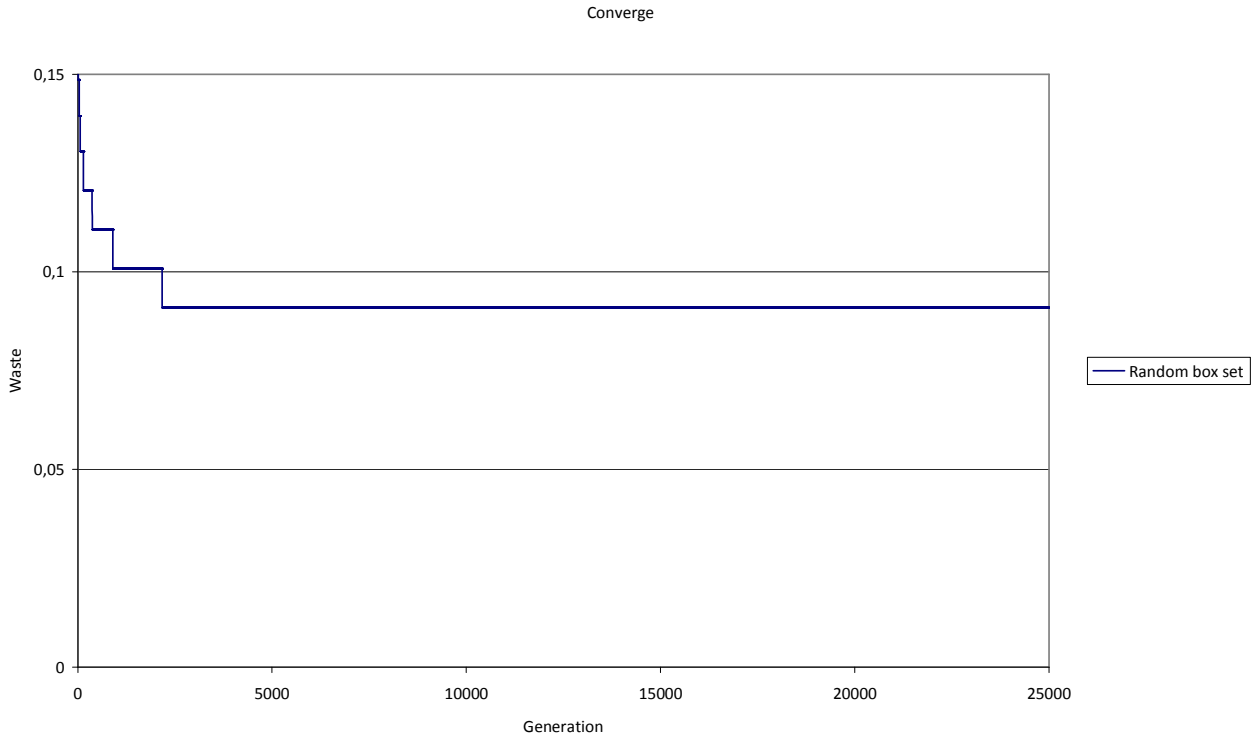


Figure 17 - Conversion

This shows without a doubt that the GA do converge towards zero, as expected. We do of cause not expect it to ever become zero. Just that it gets lower and lower over generations.

2. **How do the different parameters affect the performance of the GA (not speed)?**
   We did some runs where we made graphs with an average over 50 run for each series. We used the following basic settings.

BinWidth	=	100
GenerationSize	=	100
NumberOfGenerations	=	5000
CrossoverRate	=	0.850
MutationRate	=	0.375
GeneSwapRate	=	0.25

We then varied the three parameters one at a time, crossover rate, mutation rate and swap rate in steps of 25% and got the following three graphs.

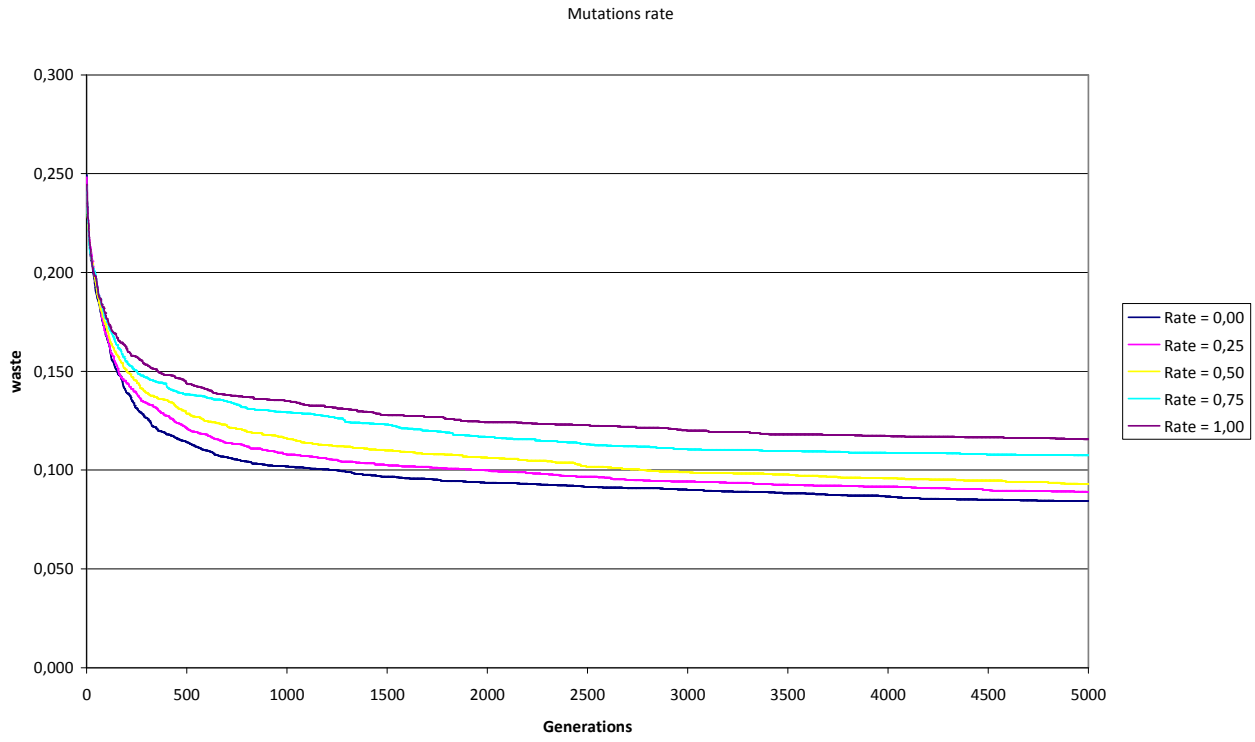


Figure 18 - Variation of mutation rate

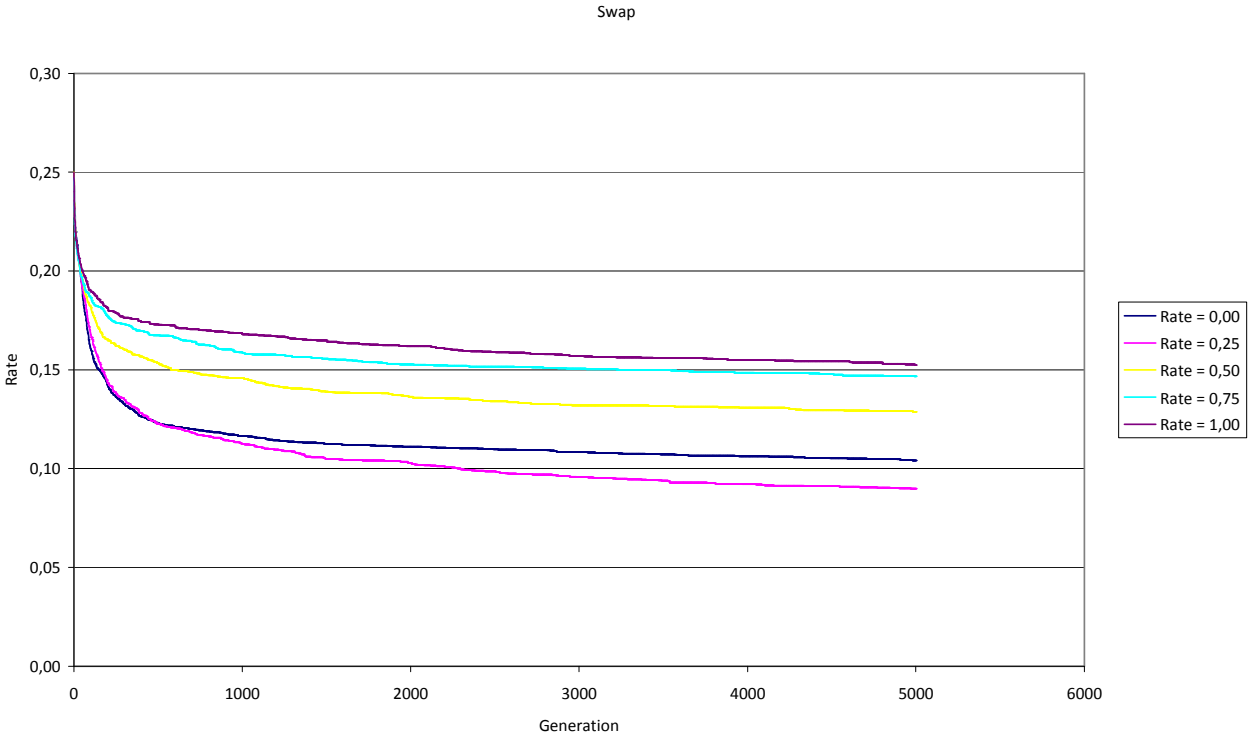


Figure 19 - Variation of swap rate

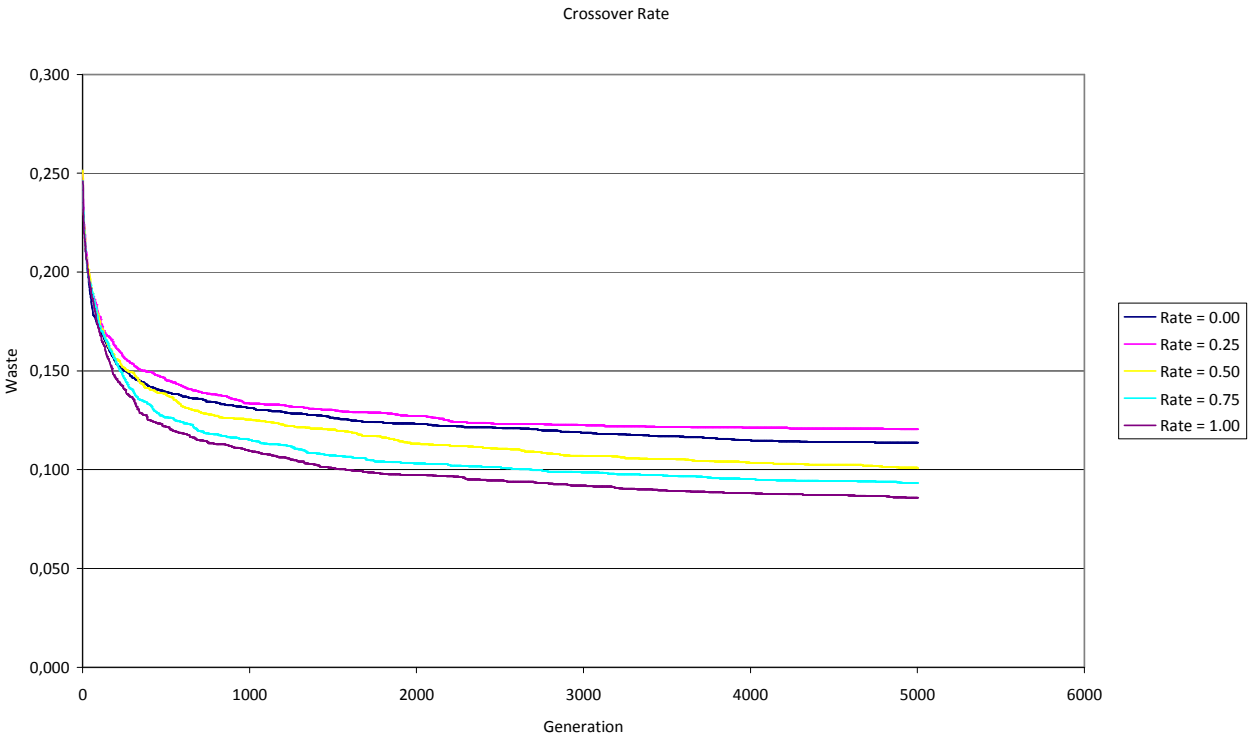


Figure 20 - Variation of crossover rate

To have the optimal parameter setting we can from these graphs see, that the mutation rate have to be between 0%-25%, the swap rate around 25% (0%-50%) and the cross over between 75%-100%. This is not necessarily the whole true as we only test in steps of 25% any setting in between could be better, but it is a good estimate.
So the setting from this point forward is

BinWidth	=	100
GenerationSize	=	100
NumberOfGenerations	=	25000
CrossoverRate	=	0.950
MutationRate	=	0.020
GeneSwapRate	=	0.250

3. **Will the GA be able to find the optimal solution, for the list of box we know is packable?**
   We made a packable set of 18 boxes as show in Figure 22 - Packable set of boxes.
   We did a graph, Figure 21 - Packable set, with an average of 50 runs which show that the GA did NOT find the optimal solution. This was not expected. We had hoped that the GA in a short amount of time would find the solution to the 18 package problem given.

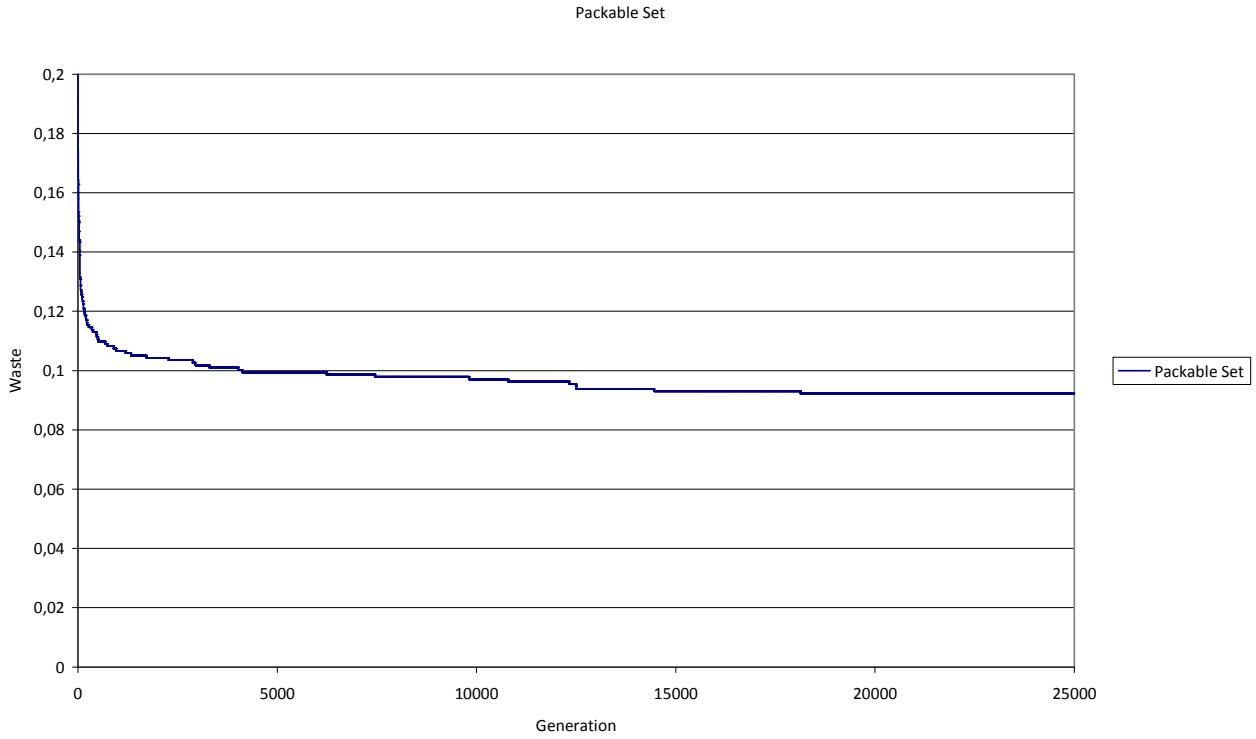


Figure 21 - Packable set

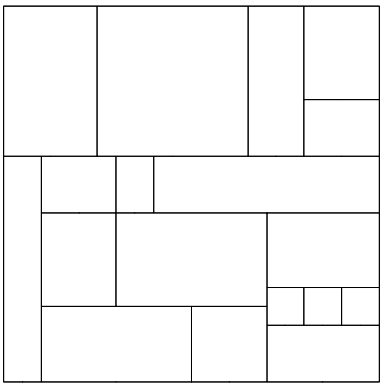


Figure 22 - Packable set of boxes

4. **How will the GA handle a huge number of boxes to pack?**
   A random set of 500 boxes was generated, and only a single run was made with the bin width of 1000. The run took about 2 hours on an Intel® Core™2 Duo Processor 6600 @ 2,40GHz with 2,00GB of ram. The implementation only ran on one core, as multithreading have not been implemented. The result shown in Figure 23 – Run with a large set of boxes, was as would be expected. The best fitness of each generation changes more often as a result of there being a lot more ways to better the packaging.

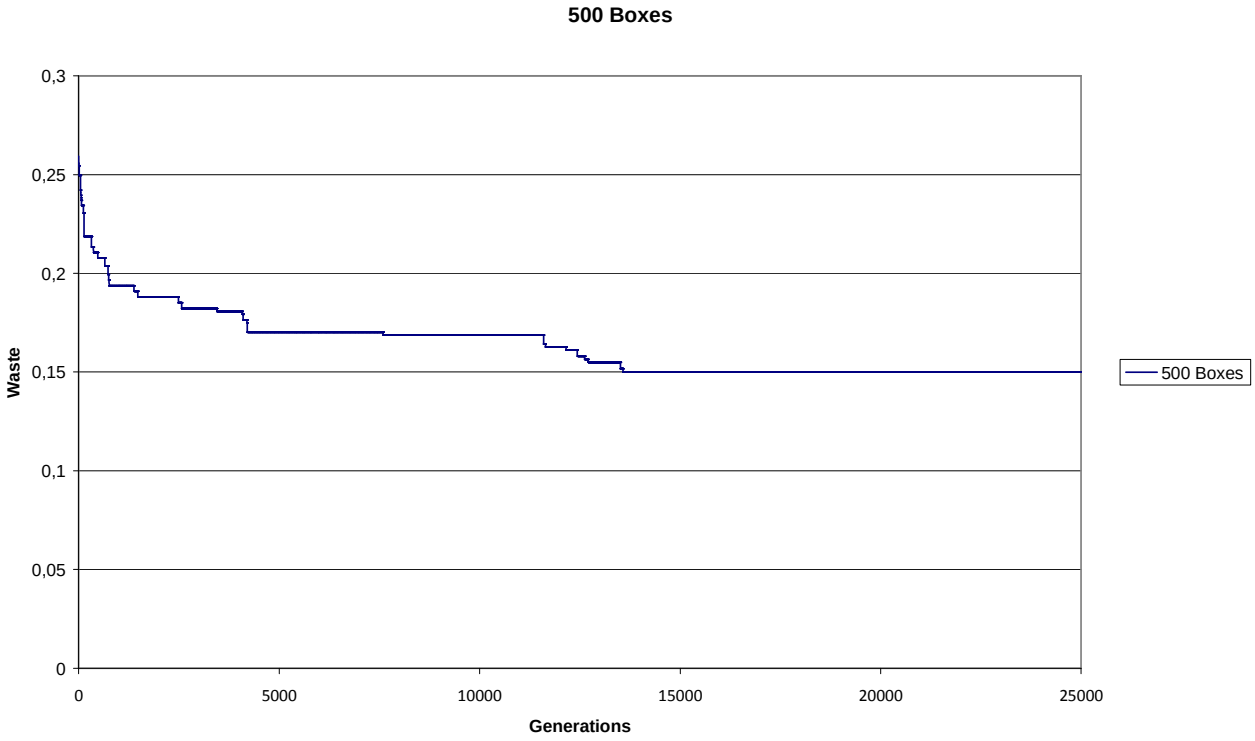


Figure 23 – Run with a large set of boxes

Discussion / Evaluation

Writing the program for the BPP helped us understand some of the problems when implementing a GA. The bin packing routine, which does the actual packing and calculation of the fitness, was by far the biggest challenge. It was hard to figure out a simple way to do a packing that was unambiguous and ensured the packages was not overlapping. The packaging is not done in a way that ensures that there are no "holes" in the bin, it is done in a way that insures that we actually get a solution. With more time and CPU time, we would have been able to figure out what combination of parameters which would be optimal, but every run (contains an average of 100 runs) takes about 30 minutes.

So where can GAs be used? If we look at the technical and business related issues, then the conclusion must be that when there is a large search area and an approximated solution is acceptable, then there is the possibility that GAs can be applied with success. But this is also the weakness of GAs, since it is an approximated solution then you have no knowledge of how good a solution the program actually will present. You can put restrictions on it so that the result must be within a certain range, but this can quickly result in a program that has to run for too long to give an acceptable result.

Even with the limited runs that we did, it is easy to see that there could be a business behind solutions like our BPP implementation. As mentioned under "Business issues", if this can help a company to improve the packing just by a few percentages then this would most likely be of interest. But the nature of GAs does make it so, that you need to carefully evaluate the complexity of both the problem and the possible solutions upfront, to insure that you get results that you can actually use outside of the computer.

If we try to take a more holistically look at the patterns in GAs, then we see that there actually is a resemblance to the way GAs works and the way we as humans sometimes acquire new knowledge.

1. First we try to solve a problem.
2. Then we evaluate the result by looking at what worked and what did not work.
3. Then we form ideas about what could be done instead to improve the result and we try again.

These three steps are in many ways very close to the cycle that is often being taught in development theories[16], so the principle of GAs is actually closer to the way we think then the model behind normal procedural driven programming. But the problem is to apply this model to a computer that relies on different logic then the cognitive one that we use; this is also why a seemingly simple problem to express can be hard to write for a computer.

Final conclusion

By looking at what GAs do best, which is to iterative improve upon a result, the conclusion must be that within search based problems sphere like the Bin Packing Problem and the Traveling Salesman Problem, there are few other tools that will be able to compete with GAs. There are often complex problems to solve

[16] http://en.wikipedia.org/wiki/Software_development_methodologies

both with doing the implementation but also with the result, but these should be solve able through control of input and size of the solution.

Glossary

BBP: Bin Packing Problem.
CPU: Central Processing Unit.
GA : Genetic Algorithm.
GP: Genetic programming
NP-hard: Non-deterministic Polynomial time hard
TSP: Traveling Salesman Problem.

Literature list.

Books:

Poli , Riccardo and Langdon, William B. and McPhee, Nicholas Freitag.
"A Field Guide to Genetic Programming" 2006
Publisher: Lulu.com

Hauot , Randy L. and Haupt, Sue Ellen
"Practical Genetic Algorithms" 2004
Publisher: Wiley-Interscience

Papers:

Cole , Nicholas, and Louis, Sushil J. and Miles, Chris
"Using a Genetic Algorithm to Tune First-Person Shooter Bots"
University of Navada

Ferreira, Cândida
"Gene Expression Programming: A New Adaptive - Algorithm for Solving Problems"
Universidade dos Açores, Portugal

Articles:

Gibbs , W. Wayt
"Programming with Primordial Ooze" – October, 1996
Publisher: Scientific American

Johson , R. Colin
"Genetic program auto-designs anlog circuits" – Issue 904, 2003
Publisher: eeTimes

Nunamaker, Jay F.
"System Development in Information System Research" – Vol 7, no 3, page 86-106 - 1991
Publisher: Journal of Management Information Systems

Links:

http://www.genetic-programming.com/humancompetitive.html

http://www.orchestrate-plan.com/orchestrate_optimizer.htm

http://blog.wired.com/music/2008/11/no-way-robot-ja.html

http://en.wikipedia.org/wiki/Kasparov#Deep_Blue.2C_1997

http://en.wikipedia.org/wiki/Technological_singularity

http://en.wikipedia.org/wiki/Software_development_methodologies